Betydelse och betydelse

Ordbetydelsedisambiguering i praktiken

Stian Rødven Eide
stian@fripost.org

Självständigt arbete i lingvistik, 15 hp
Göteborgs universitet · Institutionen för filosofi, lingvistik och vetenskapsteori

13 juni 2013

Handledare:
Torbjörn Lager, Göteborgs universitet

Sammanfattning

Ordbetydelsedisambiguering, att bestämma vilken betydelse av ett ord som gäller i ett givet sammanhang, är en av språkteknologins huvudutmaningar. Medan ordklasstaggning och informationssökning har nått mycket användbara nivåer är det fortfarande långt kvar innan vi har hittat en metod som på ett tillförlitligt sätt kan hjälpa oss att automatiskt ordbetydelsedisambiguera digital text. I denna uppsats undersöker jag vilka metoder för ordbetydelsedisambiguering som finns, hur de fungerar, samt demonstrerar hur en praktisk implementation av en sådan metod går till.

Innehåll

1 Introduktion

1.1 Inledning

Mångtydighet, eller ambiguitet, är en mycket vanligt förekommande egenskap hos naturliga språk. De 121 vanligaste substantiven i engelska har i snitt 7,8 olika betydelser, enligt den semantiska orddatabasen WordNet. [1] Medan detta inte är något problem i människans dagliga samverkan, då vi tenderar att inte ens märka den allestädes närvarande ambiguiteten när vi skriver eller läser text, är det en betydande utmaning för språkteknologin.

Forskning på ordbetydelsedisambiguering (härefter förkortat WSD efter engelskans *Word Sense Disambiguation*) har gjorts sedan de första datorerna började användas inom amerikanska universitet på slutet av 1940-talet (Locke and Booth 1955). Ändå finns det mycket arbete kvar innan datorer kommer i närheten av människans förmåga att hantera ambiguitet i en text.

Som problem för datavetenskapen har WSD beskrivits som AI-komplett, dvs. som ett problem som förutsätter en komplett förståelse av naturligt språk eller *common sense*-läsning (Agirre and Edmonds 2007). Är det verkligen så att vi inte kan lösa WSD förrän vi har en artificiell intelligens med människans språkförmåga? Svaret på denna frågan ligger naturligt nog utanför uppsatsens ramar, men förhoppningsvis kan jag bidra till en ökad insikt i problemet och de lösningar som hittills har föreslagits.

1.2 Syfte

Syftet med denna uppsats är att undersöka, förklara och demonstrera WSD som det praktiseras idag. Först ger jag en översiktlig och relativt lättfattlig införing i området, där jag beskriver de olika metoder och de mest kända algoritmer för WSD. Sedan visar jag hur man kan implementera en av dessa metoder i programmeringsspråket Python. Som exempel har jag valt Lesk-algoritmen, eftersom den är rimligt enkel både att förstå och att implementera.

1.3 Bakgrund

I korthet kan WSD beskrivas som en process för att bestämma vilken betydelse ett flertydigt ord har i en given kontext. Detta är förstås inget mål i sig, men däremot en viktig förutsättning för många språkteknologiska tillämpningar. Maskinöversättning, informationssökning och röststyrning av apparater är bara några av de mest uppenbara områden där fungerande WSD kan göra en enorm skillnad. Än så länge finns det dock inga generiska WSD-implementationer som ger tillräckligt bra resultat (Indurkhya and Damerau 2010).

En stor del av problemet är att veta var en betydelse slutar och den nästa börjar. Visst finns det ord med såpass stora skillnader i betydelsen att de kan disambigueras relativt en-

[1] http://wordnet.princeton.edu/

kelt, till exempel kan man med stor träffsäkerhet skilja mellan verb och substantiv eftersom vi har välfungerande metoder för ordklasstaggning, men detta gäller långt ifrån alla ord. Speciellt tydligt blir det när man översätter från ett språk till ett annat. Ibland är det även så att man inte är klar över nyansskillnader i ord i sitt eget språk förrän man översätter det till ett annat.

Ett vanligt förekommande exempel på ett flertydigt ord i svenskan är ”fil”. Här kommer människor troligen inte ha något problem att bestämma vilken betydelse av ordet som avses: verktyget, den mjölkbaserade produkten, datafilen eller motorvägens fil. Det framgår oftast mycket tydlig från kontexten. En dator skulle också kunna bestämma detta med ganska stor sannolikhet, om den bara har lite information om hur den skall hantera kontexten. Kommer ”äta” eller ”körde” framför ”fil” i samma mening är det lite troligt att det är verktyget som menas. Det är dock lätt att komma på meningar som skulle vara vilseledande för en datoralgoritm men som fortfarande skulle vara oproblematiskt för en människa, till exempel ”När han hade ätit halva tårtan hittade han en fil som hon tydligen hade gömt”.

Svenskans ”betydelse” är faktiskt lite vilseledande i det här sammanhanget. På engelskan skiljer man normalt mellan *meaning*, en grov och ofta uppenbar betydelseskillnad, och *sense* som är mer finkornig och utgör de stora problemen för datorer att avgöra. Detta är samtidigt illustrerande för problematiken i praktiken: I översättning till engelskan har ”betydelse” två olika betydelser, och det finns inga uppenbara markörer i den svenska texten som indikerar vilken översättning som är lämplig. I uppsatsen har jag valt att behålla ordet ”betydelse” för båda betydelserna av ”betydelse” och indikerar om jag menar en fin eller grov indelning där det är relevant.

En annan fråga är om ett WSD-system måste behandla varje ord som om det har ett begränsat antal diskreta betydelser. Ordboksvana människor är vana med detta tankesätt, men det är inte alls uppenbart. Draget till sin spets kan man säga att varje unik förekomst av ett ord har en egen betydelse och som jag skall visa finns det WSD-metoder som tar utgångspunkt i detta. Ett sådant tillvägagångssätt lär exempelvis vara användbart för maskinöversättning, där man bara behöver en indelning av ett givet ord som motsvarar ett annat språks olika översättningar av detta.

Evalueringen av WSD-system är också en viktig aspekt som bjuder på vissa utmaningar. Det är långt ifrån alla metoder som framkommit genom forskningen som har implementerats i någon praktisk tillämpning, vilket till stor del beror på att de inte är tillräckligt bra ännu. I tillägg finns det flera metoder som inte korresponderar med förehandsdefinierade ordbetydelser och därför är speciellt svåra att jämföra med. Sedan 1998 har den huvudsakliga evalueringen av WSD-system gjorts i samband med en serie internationella tävlingar, först under namnet Senseval, med enbart fokusering på ordbetydelsedisambiguering, och sedan 2007 som SemEval, när tävlingen utökades till att inbefatta andra former för semantisk analys. [2]

[2] Se http://en.wikipedia.org/wiki/SemEval för en översikt

1.4 Metod

Denna uppsats består dels av en introduktion i WSD som språkteknologiskt område, dels av en praktisk implementation av Lesk-algoritmen. Introduktionen är en sammanställning av den information som finns tillgänglig om ämnet och kan sägas vara en empirisk undersökning av litteraturen och teknologin, där huvuddelen av arbetet har bestått i att försöka tränga in i och förstå de olika metoder som finns för WSD. Tyvärr har det visat sig att litteraturen på området är något begränsad. Det finns visserligen mycket forskning på WSD, men de artiklar som publicerats är tämligen otillgängliga, både i termer av fysisk tillgänglighet och genom att de är mycket specialiserade och på så sätt ligger utanför uppsatsens ramar. Jag har därför valt att huvudsakligen förlita mig på Agirre and Edmonds (2007) som referenstext. I tillägg har jag haft mycket användning av dokumentationen för Python[3] och NLTK[4], som finns på deras respektive hemsidor, samt spridda inlägg på diverse internetforum så som Stack Overflow[5].

För implementationen har jag valt att använda programmeringsspråket Python, som är relativt enkelt för nybörjare inom programmering. Python har under senare år använts mycket inom språkteknologi, bland annat på Göteborgs Universitet, vilket i stor grad beror på tillgängligheten av modulen NLTK (Natural Language ToolKit). Denna modul innehåller flera språkteknologiska verktyg och korpusar för Python. Eftersom NLTK innehåller WordNet, ett mycket lämpligt lexikon för WSD på engelska, och att det inte finns anpassade svenska motsvarigheter, har jag valt att göra själva implementationen med engelska som exempel.

2 Olika tillvägagångssätt för WSD

Sedan WSDs början har det uppstått flera inriktningar inom området, med olika metoder för att hantera olika problem. Eftersom det inte fanns någon etablerad träningskorpus och forskningen på artificiell intelligens slog igenom först på 1980-talet var det länge bara kunskapsbaserade metoder som fanns. En stor del av forskningen på generiska WSD-system har därmed använt ett sådant tillvägagångssätt. De praktiska implementationerna som ger bäst resultat idag är dock alla korpusbaserade (Agirre and Edmonds 2007), och eftersom det är relativt stor spridning på hur systemen är uppbyggda skiljas det normalt på övervakade och oövervakade sådana. Värt att märka är att det finns olika synsätt på vad som utgör en övervakad respektive oövervakad metod. Jag har här valt att ansluta mig till Agirre and Edmonds (2007) och använder termen oövervakad enbart för metoder som inte använder någon form för taggat eller annoterat material för inlärningen. Om inlärningen däremot baseras på träningsexempel som har skapats av en människa betecknas metoden som övervakad.

[3] http://python.org/
[4] http://nltk.org/
[5] http://stackoverflow.com/

2.1 Kunskapsbaserade metoder

Medan de praktiska resultaten för kunskapsbaserade metoder för WSD tenderar vara lägre har dessa ändå en fördel framför korpusbaserade metoder i det att de hanterar alla ord i en obegränsad text. Medan en korpusbaserad metod bara fungerar med de ord som finns med i korpusen använder kunskapsbaserade metoder sig istället av olika sorters lexikon. Visserligen är även lexikon begränsade i sin omfattning, men där en inlärningskorpus innehåller ett mer eller mindre godtyckligt antal ord som råkar finnas med, eftersträvar lexikon att vara så kompletta som möjligt för det syfte de gjorts för.

2.1.1 Lesk-algoritmen

Den mest kända ordboksbaserade metoden är Lesk-algoritmen från 1986, som också var en av de första algoritmerna som kunde disambiguera alla ord i en obegränsad text (Agirre and Edmonds 2007). Lesk-algoritmen är ett typexempel som illustrerar väl hur kunskapsbaserade metoder tenderar att fungera, och den är också relativt enkel att implementera. Allt som behövs är lexikala data för varje möjlig ordbetydelse samt tillgång till den omedelbara kontexten. Det algoritmen gör är att slå upp definitioner av alla möjliga ordpar inom en viss sektion av texten, exempelvis en mening, och välja de ordbetydelser vars definitioner har ord som överlappar med varandra. Exemplet som Lesk själv valde att illustrera metoden med är ordparet *pine cone*. Båda orden har olika betydelser, men bara en av definitionerna av deras respektive betydelser innehåller ordet *tree*, och dessa betydelser väljs därmed ut som mer sannolika än de andra. Den ursprungliga formuleringen av Lesk-algoritmen presterade en korrekthet på 50-70% med Oxford Advanced Learner's Dictionary (Agirre and Edmonds 2007).

Eftersom Lesk-algoritmen skall jämföra alla möjliga kombinationer av ordens möjliga betydelser är den dock mest lämpad för disambiguering av enskilda ordpar och blir snabbt oanvändbar när meningarna blir längre. En mening som "I saw a man who is 98 years old and can still walk and tell jokes" ger till exempel 43.929.600 möjliga betydelsekombinationer, vilket innebär att den därmed inte är särskilt effektiv på längre texter.

En lösning på detta problem är att använda *simulated annealing*, en optimeringsmetod som baserar sig på sannolikhetsberäkningar. Med *simulated annealing* skapas en funktion som reflekterar alla möjliga betydelsekombinationer i en text, samt ett minimum som baserar sig på alla korrekt gissade betydelser. Algoritmen väljer sedan ut ett godtyckligt sätt med definitionskombinationer, och varje ord som finns med i någon av definitionerna får en poängsumma efter hur många förekomster det har. Summan av poängsummarna i en given kombination är textens redundans, och målet med algoritmen är att minimera redundansen. Istället för att behandla varje möjlig kombination av betydelser, går algoritmen genom iterationer av godtyckliga kombinationer och väljer i varje iteration ut den som ger minst redundans, tills den efter X antal iterationer med samma kombination inte kan hitta någon bättre. Tanken bakom detta är att komma fram till

en kombination som är bra nog för en given situation med begränsade resurser, snarare än att eftersträva någon optimal kombination som ändå är osannolik att man kommer fram till.

En annan variation är den så kallade förenklade Lesk-algoritmen som, istället för att jämföra ett ords betydelser med ett annat ords betydelser, bara behandlar ett ord i taget. Algoritmen väljer då ut den betydelse vars ord finns med i den omedelbara kontexten, alltså i själva texten och inte i andra ords definitioner. Vid flera tillfällen, bland annat WSD-tävlingen *Senseval-2*, har denna variant visat sig vara inte bara mer effektiv, utan också mer träffsäker än originalet. Den förenklade Lesk-algoritmen kan även kombineras med en annoterad korpus, vilket har visat sig vara en bra utgångspunkt för att initiera övervakade WSD-inlärningssystem. Man matchar då orden i den omedelbara kontexten både mot ett lexikon och mot exemplen i en korpus, och ger en viktad bedömning av de olika definitionerna. Algoritmen väljer så ut den definitionen som får högst total vikt. Den korpusbaserade varianten av Lesk-algoritmen räknas som den bäst presterande standardalgoritmen i jämförelse mellan övervakade WSD-inlärningssystem (Agirre and Edmonds 2007), och uppnådde en korrekthet på 69.1% i *Senseval-1*. Möjligheterna att göra variationer över Lesk-algoritmen är många, och det kommer säkerligen dyka upp flera kombinationer som hittills inte har provats. Bland annat har Banerjee och Pedersen utökat lexikondefinitionerna till att även inkludera relaterade ord i WordNet-hierarkin, för att skapa en utökad definitionsmängd (Agirre and Edmonds 2007). Tanken är att detta skall göra det enklare att hitta överlappande ord som är relevanta.

2.1.2 Semantisk släktskap

Medan Lesk-algoritmen och dess variationer baserar sig på en mycket begränsad lokal kontext, som till exempel ett ordpar eller en mening, finns det andra kunskapsbaserade tillvägagångssätt som istället försöker behandla varje ord som ingående i en global kontext. Det vill säga att man använder hela texten som underlag i disambigueringen av varje ord. De vanligaste sådana metoder går under beteckningen *semantic similarity*, eller semantisk släktskap, och baserar sig på den grundläggande premissen att ord i en diskurs måste vara relaterade i betydelse för att diskursen skall vara koherent. Tanken är att disambiguera text genom att bygga lexikala kedjor som löper från ord till ord genom hela texten.

En lexikal kedja är en inom språkteknologin mycket användbar betydelsestruktur som består av semantiskt relaterade ord i en given text. Lexikala kedjor är oberoende av den grammatiska strukturen och kan löpa över långa avstånd i texten. Av praktiska skäl är det vanligt att man baserar dessa på ord som tillhör samma ordklass, eftersom det då blir enklare att bestämma den semantiska relationen, och hittills har man nästan uteslutande använt substantiv.

Det finns ett antal metoder för att automatisk bestämma den semantiska relationen mellan ord. Av praktiska skäl finns det inte utrymme för att förklara dessa inom ramen för uppsatsen, och jag hänvisar därför till Agirre and Edmonds (2007) för ytterligare detaljer. Dock är det värt att nämna att de flesta av dessa utarbetades under 1990-talet och använder WordNet-hierarkin

som underlag för beräkningarna.

En generisk algoritm för att skapa lexikala kedjor börjar med att man väljer ut ett antal basord från texten, vars semantiska släktskap är möjligt att beräkna utifrån den valda metoden. För varje basord och för varje betydelse som detta ordet kan ha, placeras betydelsen i varje kedja där den enligt beräkningsmetoden får vara. Om den inte passar in i någon existerande kedja skapas en ny. Alla kedjor som överstiger en viss tröskel väljs sedan ut för en slutlig jämförelse.

Metoder för att disambiguera genom semantisk släktskap lider dock i grunden av samma problem som originalversionen av Lesk-algoritmen; det blir snabbt ohanterligt många möjliga kombinationer att jämföra. För att begränsa mängden av betydelsemöjligheter har därför olika lösningar provats, bland annat att utgå ifrån den lokala kontexten i första hand, och på så sätt ge en första bedömning av vilka betydelser som är mer sannolika, eller att introducera syntaktiska beroenderelationer, där syntaxen automatiskt får utesluta vissa semantiska relationsmöjligheter.

2.1.3 Urval genom företräde

Bland de första algoritmerna för WSD var flera baserade på urval genom företräde (eng. *selectional preference*), alltså en *common sense*-läsning av text där man väljer ut en betydelse utifrån klasser av begrepp. ”Äta-mat” och ”dricka-vätska” är typiska exempel på sådana begrepp. I meningen ”Eva drack ett glas rött” skulle man därmed lätt sluta sig till att ordet ”rött” hänvisar till rött vin, med hjälp av en regel som säger att verbet ”dricka” föredrar vätska och en av betydelserna för ”rött” är ett substantiv med egenskapen ”vätska”. Detta liknar det sätt på vilket människor disambiguerar ord, men där människan har en till synes obegränsad mängd sådana begrepp som ligger till grund för hennes *common sense* är utmaningen med ett sådant tillvägagångssätt för WSD att varje begreppsklass måste beskrivas utförligt med en regel. Det är därmed lätt att hamna i en beroendecirkel: för att lära in begreppsliga restriktionsregler krävs kännedom om relationen mellan olika ords betydelser och tvärt om. För att bryta denna cirkel behövs i princip en stor uppsättning färdigdefinierade begreppsklasser.

2.1.4 Heuristiska metoder

En av de enklaste metoderna för att disambiguera ord är att skapa regler utifrån språkliga egenskaper som kan observeras i större texter. Sådana metoder utgår ifrån heuristiska sannolikhetsberäkningar, och försöker på så sätt bara att göra rimliga antaganden om vad som kan förväntas. Heuristiska tillvägagångssätt för WSD har även visat sig vara tämligen precisa, men kräver att texten som skall disambigueras är hyfsat enhetlig, och kommer därför inte att fungera väl för alla sorters text.

Det finns i huvudsak tre heuristiska urvalsmetoder som har använts inom WSD: 1) Den oftast förekommande betydelsen, 2) en betydelse per diskurs och 3) en betydelse per kollokation. Urval genom den oftast förekommande betydelsen baserar sig på Zipfs distributionanalyser,

som säger att en specifik betydelse av ett givet ord kommer vara dominerande inom alla texter på ett givet språk, med en snabbt avtagande kurva för alla andra möjliga betydelser (Bird, Klein and Loper 2009). Det är därmed rimligt att anta att de allra flesta förekomster av ordet i en given text kommer ha denna betydelse, och att utgå ifrån denna som ett grundantagande kommer ofta att ge en tämligen korrekt disambiguering. Denna metod är relativt enkel att implementera, men har två uppenbara brister. Den ena är att det behövs en översikt över olika ordbetydelsers frekvensdistribution, vilket bara existerar för ett antal språk. Den andra är det att texter inom en specifik domän inte alltid följer samma distribution som språket i sig.

Mer användbart för domänspecifika texter är därmed att istället utgå ifrån en betydelse per diskurs. Det innebär i praktiken att disambiguering av en given text måste sättas igång med en förutsättning om vilka betydelser som är mest relevanta och sannolika, till exempel en domänspecifik samling frekvensdistributioner. Försök med denna metod har dock visat att ungefär 33% av ord används i flera olika betydelser i en given diskurs, och det är därmed svårt att komma över en disambigueringsnivå på mer än 70%. Som Yarowsky har visat (Agirre and Edmonds 2007) är denna metod däremot mycket användbar för att sätta igång korpusbaserade metoder, och förbättrade hans *bootstrapping*-algoritm från 90,6% till 96,5%. [6]

En annan heuristisk variant som introducerades av Yarowsky är utgångspunkten att ett ords betydelse nästan alltid kommer vara den samma i en given kollokation. Detta har visat sig stämma bra för indelning i grova betydelseskillnader, men fungerar mindre bra för den sortens finindelning som har gjorts i bland annat WordNet (experiment gjord av Martinez & Agirre, 2000, se Agirre & Edmonds, 2007). Samtidigt är det så att metoden fungerar lika bra i olika korpusar, betydelsen inom kollokationer tenderar vara den samma, men däremot finns det inte särskilt bra överlapp mellan kollokationer inom olika korpusar, så att träna in regler för kollokationer kommer troligen inte känna igen de flesta kollokationer i en annan korpus.

2.2 Övervakade korpusbaserade metoder

Korpusbaserade metoder som baserar sig på maskininlärning av regler har de senaste 10 åren visat sig vara mycket effektiva för automatisk WSD (Indurkhya and Damerau 2010). Inlärningen görs från en träningskorpus, och om metoden betecknas som övervakad innebär det i princip att korpusen har taggats manuellt. Genom processen lär sig programmet vilka betydelser som gäller för ett givet ord i en given kontext och skapar regler för vilken disambiguering som skall gälla i en given situation. Sådana metoder fungerar relativt bra för att disambiguera text som liknar de texter den har tränats med och är därför ett naturligt val inom domänspecifika områden.

Huvudproblemet för övervakade korpusbaserade metoder är det som kallas kunskapsflaskhalsen (eng. *the knowledge aquisition bottleneck*) (Agirre and Edmonds 2007). Eftersom de baserar sig på annoterad text krävs det en enorm mängd träningstext för att dessa metoder skall

[6] Yarowsky's Bootstrapping Algorithm förklaras närmare i sektion 2.4.1

fungera effektivt med vilken som helst annan text. Det skulle till exempel inte vara så lämpligt att använda ett WSD-system som har tränats på texter inom medicinsk forskning för att översätta en kokbok.

Språkteknologiska utmaningar som maskinöversättning, ordklasstaggning och WSD matchar maskininlärning perfekt, eftersom just klassifiering kan sägas vara en av maskininlärningens grundfunktioner (Indurkhya and Damerau 2010). Det är dock inte alltid lätt att reducera NLP (från engelskans *Natural Language Processing*) till ren klassifiering, särskilt inte när man har att göra med mer komplexa hierarkiska strukturer, så som syntaxträd inom vissa modeller.

De regler som övervakade inlärningsmetoder typiskt skapar är i huvudsak baserade på den omedelbara kontexten: exempelvis vilka andra ord som överhuvudtaget finns med i närheten, vilka vanligt förekommande bigram och trigram som ett ord förekommer i, och sträcker sig sällan utanför meningen. Ett klassifikationsschema består av ett antal sådana regler, med en tydlig precedens där varje regel har ett sannolikhetsvärde som säger hur mycket vikt den har gentemot andra regler. Det innebär dock att även om regeln med högst precendens föredrar en betydelse har de andra reglerna gemensamt mer vikt om det enligt dem är en annan betydelse som bör väljas.

Bristen på träningskorpusar är det största problemet för övervakade metoder och det har det lagts mycket arbete på att bygga upp sådana. I tillägg till engelskspråkliga Semcor och DSO, som båda har taggats med WordNet-betydelser, har det byggts upp flera mindre korpusar på olika språk i samband med *Senseval*-tävlingarna. En ytterligare korpus, The Open Mind Word Expert Project, påbörjades i 2002 och baserar sig på hjälp från volontärer som hjälper till med taggningen via webben i form av ett dataspel. [7]

Grunden för ett lyckat övervakat korpusbaserat WSD-system är förstås träningsexemplen. Det är nämligen dessa som ligger till grund för de regler som lärs in, och det är också där som största risken för disambigueringsfel ligger (Agirre and Edmonds 2007) Även en bra algoritm för att inducera regler kommer att lära in felaktiga regler om den inte tränas på fullgoda exempel, dvs exempel som i största möjliga grad tillåter alla möjliga tolkningar och inga ogiltiga tolkningar. Själva kodningen av exemplen och algoritmerna som tolkar dessa behandlar varje ordbetydelse som punkter i ett *n*-dimensionellt rum, där n är antalet egenskaper som en betydelse kan ha. Egenskaperna bestäms genom ett antal metoder och inkluderar både syntaktiska regler baserade på ordklass och semantiska regler för att känna igen semantiska indikatorer i exempelvis *n*-gram. För övervakade metoder är detta oftast det första steget, en förbearbetning av träningsexemplen som delar upp texten i lämpliga bitar och bygger upp ett sätt med egenskaper kring de olika möjliga ordbetydelserna. Anledningen att detta görs för varje inlärning och att man inte använder någon generisk databas över vektoriserade ordbetydelser är att det finns stora skillnader mellan hur maskininlärningsalgoritmerna hanterar olika klasser av egenskaper. Alla algoritmer har sina brister, och att utelämna mindre kompatibla egenskaper från

[7] http://www.sussex.ac.uk/Users/robk/OpenMind/template/english/index.html

träningsexemplen kommer därmed bara att förbättra det slutliga resultatet.

Här följer en kortfattad översikt över de olika tillvägagångssätt som vanligtvis görs inom övervakad korpusbaserad WSD. Den grundläggande skillnaden är vilka induktionsprinciper som ligger till grund för inlärningen.

2.2.1 Statistiska sannolikhetsmetoder

En av de enklaste och vanligaste algoritmerna för maskininlärning av språkregler är Naive Bayes-algoritmen, som baserar sig på statistisk sannolikhet. Den är ett typiskt exempel på en statistisk metod för WSD, och konstruerar en uppsättning sannolikhetsparameter för varje träningsexempel. Exemplen sorteras därefter i kategorier allt efter vad som kan maximera sannolikhetsvärdet för en korrekt disambiguering. Exempelvis kan man rimligen anta att verbs egenskaper i högre grad placeras i syntaktiska kategorier, eftersom det ofta finns en syntaktisk relation mellan verb och vilka andra ordklasser som finns med i den omedelbara kontexten, medan de flesta substantiv kanske passar bättre i semantiska kategorier. Det finns dock ingen principiell indelning enligt ordklasser, utan den görs enbart mot bakgrund av statistisk resultatmaximering.

2.2.2 Likhetsprincipen

Metoder baserade på likhetsprincipen utgår helt enkelt ifrån en modell, vanligtvis så kallad vektorrumsmodell, som säger hur lika två exempel är varandra, och hur nära ett stycke text som skall disambigueras ligger exemplen. Beräkningen inleds normalt med skapandet av en prototyp av vektoriserade egenskaper för varje ordbetydelse och sedan skapas en databas med olika exempel som grupperas efter likhet. Den vanligaste algoritmen för detta är *k*NN där *k* är en siffra som säger hur många exempel ett ord skall likna och NN står för *Nearest Neighbours*. *k*NN-algoritmen anses vara mycket användbar, bland annat eftersom den inte gör någon generalisering när den skapar exempel och på så sätt inte bortser från undantag, vilket det tenderar att finnas många av i naturliga språk.

2.2.3 Diskrimineringsregler

Något enklare än vektormodeller är att bygga diskrimineringsregler, som är uppsättningar av beslutslistor eller -träd. I sin grundläggande form kan en beslutslista beskrivas som en viktad *if-then-else*-sats, det vill säga som en lista med regler i en viss ordning där en regel bara används om ett ord inte mötte användningskriterierna för de föregående reglerna. Regler för speciella undantag kommer normalt i början, och till sist kommer en standardregel som bara gäller om inga andra regler gör det. Vikten som beledsagar varje regel utvinns från träningsexemplen och säger helt enkelt hur sannolikt det är att regeln är korrekt om villkoren uppfylls.

2.2.4 Regelkombinationsmetoder

Eftersom de flesta metoder har brister är det naturligt att kombinera dessa i olika konstellationer för att jämna ut deras svagheter och, i bästa fall, välja regler från den metod som är mest lämpad för en given situation. Flera försök har gjorts med detta, och en av de mest lyckade är AdaBoost-algoritmen. Denna kombinerar flera klassifierare, som t.ex. diskrimineringsregler och statistiska regler, med förtroendevärden som specifierar hur tillförlitlig en given klassifierare är i ett givet fall. Att använda regelkombinationer har visat sig vara mer träffsäkert än enskilda metoder, men än så länge bara marginellt (Agirre and Edmonds 2007). Det är därmed inte givet att det lönar sig i praktiken eftersom det krävs avsevärt mer resurser att köra flera inlärningsalgoritmer parallellt.

2.2.5 Linjära klassifierare

Linjära (binära) klassifierare påminner om likhetsmetoder i det att varje betydelse är representerad av vektorer i ett mångdimensionellt rum. Den stora skillnaden är att betydelserna väljs ut efter närheten till en prototyp som gradvis byggs upp istället för basera urvalet på jämförelse av exempel. Linjära klassifierare har använts mycket inom informationssökning (vanligtvis betecknat IR efter engelskans *Information Retrieval*) men relativt lite forskning har gjorts på rena linjära modeller.

2.2.6 Kärnbaserade metoder

Kärnbaserade metoder kan ses som en utvidgning av linjära klassifieringsmetoder där även icke-linjära funktioner tas i bruk. Med hjälp av så kallade kärnfunktioner klarar man att begränsa funktionernas omfång till relevant data, något som gör dessa metoderna tämligen effektiva. Kärnbaserade metoder har fått mycket spridning de senaste åren och låg bakom flera av de bästa resultaten under Senseval-3.

2.3 Oövervakade korpusbaserade metoder

I en oövervakad metod har inte träningskorpusen taggats. Istället för att använda taggar väljer sådana metoder en ordbetydelse baserad på information från rå text. Denna process kan vara antingen typ-baserad, i den meningen att den identifierar en grupp av relaterade ord, eller förekomst-baserad, att den väljer bland ordbetydelser som förekommer i olika sammanhang. Fördelen med oövervakade metoder är att de undviker kunskapsflaskhalsen, men samtidigt kräver de många iterationer av träning och stora mängder text för att prestera i närheten av det övervakade metoder gör.

Inom de oövervakade metoderna finns det huvudsakligen två alternativa riktningar. Den ena är de distributionella metoderna, som gör distinktioner i ordbetydelser baserat på antagandet

att ord som dyker upp i liknande sammanhang också har liknande betydelser. Den andra är de översättningsekvivalenta metoderna, vilka är baserade på parallellkorpusar och identifierar översättningar av ett ord till ett målspråk som beror på en viss betydelse av ordet i källspråket.

2.3.1 Distributionella metoder

Distributionella metoder är i princip en maskinell ekvivalens till det arbete en lexikograf gör. Det första steget är att hitta förekomster av ett givet ord i olika sammanhang. Kontexterna som identifieras delas sedan in i olika kluster, baserat på hur de liknar varandra (distributionell karakteristik). Klustren som identifieras motsvarar de olika betydelserna av ordet och bildar utgångspunkten för definitionerna. Det finns på så sätt inget förutbestämt om indelningen, utan metoden, på samma sätt som lexikografen, bestämmer hur många olika betydelser ett ord har utifrån materialet. Det andra steget är att på något sätt studera de olika klustren och bestämma vilka definitioner som skall tillämpas. Medan detta är något som mänskliga lexikografer är bra på, har maskinella tillämpningar fortfarande inte möjlighet att prestera något motsvarande (Agirre and Edmonds 2007).

En lösning på problemet med att tagga klustren och bestämma själva definitionerna är att ta utgångspunkt i en förhandsdefinierad lista av betydelser, som t.ex. WordNet. Detta kan dock inte anses som en oövervakad metod, eftersom den baserar sig på manuellt taggad information, och står därmed inför samma utmaningar som kunskapsbaserade och övervakade metoder: Man är begränsad till de få betydelselistor som finns, på de språk som de finns för, och samtidigt överlämnad till en given indelning av betydelser som kanske inte är lämpad för alla situationer.

En annan och vanligare lösning är att helt enkelt utvinna definitionen från kontexten. Det är här som typ- och förekomstbaserade metoder skiljer sig åt.

2.3.2 Typbaserade metoder

I typbaserade metoder genereras en uppsättning ordtyper som tenderar att figurera i samma sorts kontext. Denna lista blir då själva definitionen av varje ordbetydelse. Exempelvis skulle ordet ”fil” kunna få en definition som är [fil, trafik, bil, väg] och en annan som är [fil, frukost, müsli, äta]. Sådana metoder kallas typbaserade därför att klustren inte innehåller information om individuella ords förekomster, utan enbart om relationen mellan orden.

Målet med typbaserade metoder är att på så sätt fånga ords kontextuella likhet. För varje ord som skall disambigueras byggs det upp en profil som representerar deras möjliga kontexter. Mer konkret skapas en matris som innehåller i första hand samförekomster av ord, både bigram och oordnade sådana. I tillägg är det vanligt att inkludera en eller flera associationsmätningar, till exempel den så kallade *log-likelyhood ratio*, som indikerar hur sannolikt det är att en given samförekomst av två ord är ett bigram och inte en slumpmässig samförekomst. När profiler har skapats för alla relevanta ord i texten bildas så definitionslistorna, baserad på vilka profiler

som liknar tillräckligt på varandra; eftersom de finns i relaterade kontexter antas det att de har relaterade betydelser.

Det finns huvudsakligen tre algoritmer som är vanliga inom typbaserad WSD (Agirre and Edmonds 2007): Latent Semantic Analysis (LSA), Hyperspace Analogue to Language (HAL) och Clustering by Committee (CBC). Medan alla tre representerar samförekomster av ord som vektorer i en mångdimensionell rymd, använder LSA och HAL enbart en vektor för att representera varje ord och kan därmed inte hantera polysemi i texten. Målet med dessa algoritmer är snarare att bestämma vilka ord som är relaterade, samt vilken betydelse som dominerar i en given text. I CBC identifieras däremot flera kluster per ord, vilket möjliggör disambiguering på en polysemisk nivå. För att kunna göra detta analyseras kontexten även syntaktiskt. CBC är därmed inte helt oövervakad som LSA och HAL, eftersom den även använder en manuellt skapad syntaktisk parser, men har i övrigt många likheter med dem.

Eftersom både LSA och HAL inte skiljer på polysemi anses typbaserade metoder i allmänhet vara bäst lämpade för disambiguering inom domänspecifika områden, där en given betydelse av ett ord anses dominera. Det är dock allmänt vedertaget att även en algoritm som konsekvent väljer ut den oftast förekommande betydelsen är en mycket bra standard för WSD (Agirre and Edmonds 2007). Typbaserade metoder lyckas speciellt väl med detta inom domänspecifika texter eftersom de inte gör någon bedömning av vad som är vanligt i språket i allmänhet, utan enbart utgår ifrån texten i fråga.

2.3.3 Förekomstbaserade metoder

I motsats till typbaserade metoder, klustrar förekomstbaserade metoder ihop hela kontexter. Att de kallas förekomstbaserade beror just på att varje kontext, och därmed varje förekomst av varje ord, bevaras i ett kluster. Dessa metoder taggar dock inte klustren och det krävs att en människa går in och gör det. Ofta är det också så att en viss grad av klustring av ordförekomster kan vara nödvändig innan typer kan hittas genom typbaserade algoritmer.

Två av de tidigaste förekomstbaserade tillvägagångssätten var Context Group Discrimination (CGD) och en implementation av McQuitty's Similarity Analysis (MSA) (Agirre and Edmonds 2007).

CGD utgår ifrån LSA-algoritmen som redan använts i typbaserad diskriminering, men lägger till ytterligare två vektorrymder. Med utgångspunkt i ordvektorerna, som är samförekomstmatriser liknande de som görs med LSA och HAL, skapas först en kontextvektor. Denna konstrueras helt enkelt genom att slå ihop ordvektorerna från alla ord i en given kontext och räkna ut ett medelvärde, som då bildar en ny vektor. Med denna vektor får man en representation av andra ordningens samförekomst, dvs. ord som förekommer tillsammans med ett givet samma andra ord. Exempelvis förekommer det engelska ordet ”crude” tillsammans med både ”oil” och ”sugar”. ”Oil” och ”sugar” har då en andra ordningens samförekomst. Huvudargumentet för att använda dessa är att bigram inte förekommer tillräckligt ofta, och det är därmed större chans för

att upptäcka semantiska relationer genom att även inkludera andra ordningens samförekomster (Agirre and Edmonds 2007). När alla kontextvektorerna har skapats används en klusteralgoritm som räknar ut likheten mellan olika kontexter och resultatet är en tredje vektorrymd bestående av betydelsevektorer. Varje betydelsevektor representerar en enskild betydelse av ett givet ord.

MSA bygger också upp vektorer, men dessa är däremot mindre baserad på samförekomster än CGD och består istället av flera egenskaper, både hos ordet som skall disambigueras och orden i den övriga kontexten. De data som MSA hämtar in, i tillägg till samförekomster, är den morfologiska formen av målordet, ordklass för närliggande ord samt övriga kollokationer. Själva algoritmen använder sedan denna data för att diskriminera mellan olika ordbetydelser. Det vanliga med sådana algoritmer är att de först utgår ifrån att alla förekomster representerar en unik betydelse, och kör sedan flera iterationer över texten. För varje iteration slås olika kluster ihop, baserad på likhetsanalysen som är inbakad i algoritmen, tills programmet når en satt gräns. När gränsen nås sitter man kvar med ett antal olika kluster, som vart och ett representerar en given betydelse för ett ord.

Huvudutmaningen för förekomstbaserade metoder är att de inte taggar de olika kluster som byggs upp. För att göra det måste man antingen manuellt gå igenom resultaten, eller så får man kombinera dem med något redan existerande lexikon, vilket är mycket svårare att koppla till kluster än till enskilda ord. Om man skall använda lexikon är det därför vanligt att kombinera typ- och förekomstbaserade algoritmer. Båda tar iallafall bort poängen med en oövervakad metod. En fördel med denna brist på koppling till existerande lexikon är dock att man slipper dilemmat med fördefinierade betydelsedistinktioner, som inte alltid är de bästa för en given situation. Det innebär att även om det är svårt att evaluera förekomstbaserade metoder och att de inte är det optimala valet för generisk WSD, så kan det vara mycket frigörande att inte vara bunden till redan existerande orddistinktioner. Oövervakade metoder är på så sätt mycket användbara i specifika situationer där man inte har något direkt behov av mänskligt definierade distinktioner, till exempel inom informationssökning eller maskinöversättning.

2.3.4 Översättningsekvivalens

Oövervakade korpusbaserade metoder har även använts inom maskinöversättning, och baseras då på översättningsekvivalens. Detta innebär att man utgår ifrån en parallellkorpus, en samling med texter som finns tillgängliga på flera språk. Med hjälp av denna tränas så in regler som specifierar hur det skall översättas från ett språk till ett annat. Texterna måste dock vara översatta på ett sådant sätt att det är möjligt att länka dem. Det betyder i praktiken att till exempel en roman inte skulle vara särskilt lämpligt material, eftersom översättning av skönlitterära texter nödvändigtvis innebär en hel del omskrivning för att behålla den litterära kvaliteten. En stor utmaning för maskinöversättning baserad på översättningsekvivalens är därför att det inte finns tillräckligt med parallella texter att använda som inlärningsmaterial.

Själva länkningssprocessen är normalt tredelad. Först görs en länkning på meningsnivå, vil-

ket det idag finns pålitliga metoder för att göra. Att länka själva orden är dock betydligt svårare, till en sådan grad att det har diskuterats huruvida WSD alls är lämplig för maskinöversättning. Under 2000-talet har det dock gjorts lyckade försök på att länka parallellkorpusar ord för ord för ett antal språk (Agirre and Edmonds 2007). Efter länkningen byggs så en träningskorpus där lexikala och syntaktiska egenskaper för de motsvarande orden ingår. Utifrån detta skapas sedan översättningsregler, antigen helt maskinellt eller i en kombination med övervakade inlärningsalgoritmer.

2.4 Kombinationer

På grund av bristen på träningsdata är det också vanligt att använda en kombination av övervakade och oövervakade metoder. Man börjar då med antigen en mindre träningskorpus eller en begränsad uppsättning regler och skapar en större databas från detta, en klassifierare. Utifrån klassifieraren bygger programmet upp en större mängd data där vissa betydelser markeras som säkra och andra som mindre säkra. Dessa kan återigen matas in i klassifieraren genom flera iterationer tills hela korpusen har taggats med säkra träffar.

2.4.1 Bootstrapping

En speciellt lyckad kombinationsmetod är den så kallade Yarowsky Bootstrapping Algorithm (YBA). Den betecknas normalt som en semi-övervakad metod eftersom den börjar med ett litet sätt av taggade exempel och utökar sedan omfånget till att gälla fler och fler otaggade exempel. Denna metod illustrerar mycket väl det tillvägagångssätt som löser *bootstrapping*-problemet, alltså hur man kan bygga upp det nödvändiga initiala underlag som krävs för att en automatiserad WSD-algoritm skall kunna disambiguera i enlighet med fördefinierade betydelsedistinktioner gjorde av människor.

YBA är också ett typexempel på en iterativ och inkrementell algoritm. Utgångspunkten är en relativt liten grupp exempel som har taggats, en större grupp exempel som inte har taggats och en inlärningsalgoritm som skapar beslutslistor. För varje iteration lär sig algoritmen några nya betydelsedistinktioner. De den inte är säker på, dvs. de som inte når en satt nivå av träffsäkerhet, förkastas tillsvidare. Under nästa iteration har algoritmen därmed mer data att basera analysen på, och träffsäkerheten för de olösta ambiguiteterna ökar successivt. För att beräkna träffsäkerheten används flera av de algoritmer som ingår i kunskapsbaserade metoder, speciellt heuristiska algoritmer som ”en betydelse per kollokation”.

3 Implementation av Lesk-algoritmen

Som tidigare nämnt är Lesk-algoritmen en av de första och mest kända kunskapsbaserade algoritmerna för WSD. Eftersom den även är relativt enkel att implementera är det ett rimligt val av

algoritm för att undersöka och demonstrera hur WSD kan göras i praktiken. Själva algoritmen är tämligen grundligt beskriven i sektion 2.1.1, så jag tänker inte återupprepa något om den här. Min källkod för programmet återges i sin helhet i sektion 3.2.

3.1 Beskrivning av arbetet

Arbetet med att implementera en algoritm börjar rimligtvis med att studera och försöka förstå algoritmen. Av alla algoritmer som jag har berört i uppsatsen är Lesk den som jag inte bara anser vara enklast att implementera, men också den som bäst illustrerar hur WSD *kan* fungera. Jag har kommenterat min kod grundligt och hoppas den skall vara tydlig och informativ även för de som inte har någon erfarenhet av programmering.

Eftersom min egen bakgrund som programmerare också är mycket begränsad har jag förhållit mig till det språk och de verktyg som jag har erfarenhet av. Programmeringsspråket Python och modulen NLTK var därför ett naturligt val. I tillägg till att vara särskilt lämpat för nybörjare har Python även en stor användarbas inom språkteknologi, och dess läsbarhet gör det utmärkt som pedagogiskt verktyg för att demonstrera en implementation som denna.

NLTK innehåller en stor mängd korpusar och verktyg för att hantera dessa. Inbyggd finns bland annat klassiska romaner, tidningsartiklar, politiska tal, tekniska texter, samt den semantiska databasen WordNet. WordNet är den idag största och mest använda semantiska databasen för språkteknologisk forskning och innehåller, i tillägg till betydelsedistinktioner och definitioner på dessa, flera semantiska relationer så som hyperonym, hyponym, holonym, antonym, samt verktyg för att beräkna likhet mellan ordpar. Flera av metoderna beskrivna i denna uppsats har huvudsakligen testats med WordNet.

Idealiskt vore naturligtvis att disambiguera på svenska, men på grund av bristen på väldokumenterade databaser och verktyg bestämde jag mig för att använda WordNet och därmed engelska. En annan fördel med att implementera algoritmen med engelska som mål är det faktum att man underlättar för andra som vill prova koden. Genom att använda ASCII för källkoden, vilket exkluderar alla accentuerade tecken med mera, är det nämligen oproblematiskt att köra samma kod på olika datorer och operativsystem. I dagsläget är det huvudsakligen bara GNU/Linux-system som använder unicode, en teckenkodningstabell som stödjer de allra flesta tecken som förekommer i de allra flesta språk, som standard. För att bevara ASCII-enkodningen har jag även använt engelska kommentarer i koden. För att köra koden behövs Python, med NLTK installerad. Från Pythons hemsida finns utförliga beskrivningar för hur man installerar programmet i ett antal operativsystem.

Jag valde att implementera den ursprungliga varianten av Lesk-algoritmen. Anledningen till detta är först och främst för att den är enkel att testa. Istället för en hel träningstext behöver den bara två ord, ett ordpar, som den slår upp definitionerna för. Den förenklade Lesk-algoritmen har visserligen visat sig att vara mer effektiv, men för denna behöver man en större text som under-

lag. Det lär dock vara relativt enkelt att modifiera koden för att anpassa den till den förenklade Lesk-algoritmen.

Själva arbetet med implementeringen innebar i stor del att formulera proceduren inom Pythons ramar. Programmet börjar med att fråga användaren efter två ord. Dessa måste nödvändigtvis finnas med i en engelsk ordlista för att kunna disambigueras. Det första algoritmen gör är att slå upp orden i WordNet och hämta alla olika betydelser dessa kan ha. Varje betydelse för varje ord sparas så internt i programmet, i datatypen *dictionary* så att man kan lägga till ett värde för varje betydelse. Detta värdet sätts så initialt till 0 och anger hur många förekomster av ett givet ord från en definition som finns med i en definition för det andra ordet i ordparet. Huvuddelen av programmet, och det som krävs mest resurser, är en loop som tokeniserar varje definition och räknar förekomster av ord i dem. Varje definition för varje betydelse jämförs mellan båda orden, och för varje gång ett ord förekommer i en definition för båda orden får dessa betydelserna en extra poäng tillagt till sitt värde. När alla definitioner gåtts genom plockar programmet ut den betydelse för varje av de två orden som har fått flest poäng och presenterar resultatet för användaren. Om flera betydelser har samma högsta poängsumma ges en lista med dem alla. Om inga betydelser får poäng alls får användaren veta detta. För ytterliga detaljer hänvisar jag till källkoden i Appendix A.

4 Slutsatser

Under sammanställningen av de olika metoder som finns för WSD var det uppenbart att de alla har stora brister, vilket också är tydligt från de resultat som framkommit under *Senseval*-tävlingarna. Det största problemet lär därmed vara att ta fram en metod som på ett tillförlitligt sätt kan fungera som en generisk WSD-implementation och hantera alla sorters text och användningsområden. Däremot kan vissa av bristerna fungera som en fördel för vissa tillämpningar, vilket exempelvis är fallet med oövervakade metoder för maskinöversättning, eftersom dessa då inte behöver korrespondera med fördefinierade betydelsedistinktioner. Medan kunskapsbaserade metoder har den största potentialen för att göra korrekta bedömningar är det också så att dessa i störst grad beror på regler definierade av människor och därmed lider av kunskapsflaskhalsen. Övervakade metoder är de som idag presterar bäst resultat i de evalueringar som finns, men även för dessa är bristen på taggade exempel att lära ifrån påfallande. Oövervakade metoder saknar denna begränsning men det är därmed också svårt att relatera dem till redan existerande betydelsedistinktioner. Det jag ser för mig som den bästa lösningen är därför att kombinera metoder från alla tre områden, så som Yarowskys Bootstrapping Algorithm. Man kan då gradvis bygga upp en kunskapsdatabas av regler med hjälp av oövervakade metoder som initieras genom övervakad maskininlärning.

När det gäller min implementation upplevde jag att valet av algoritm och programmeringsspråk var optimala för syftet. Pythons enkelhet och läsbarhet kombinerat med den kraftfulla

modulen NLTK gjorde att utmaningen med att implementera Lesk-algoritmen gick tämligen smidigt. Utan större problem fick jag programmet att bete sig i enlighet med mina förväntningar. Det som visade sig vara det största problemet var däremot WordNet. Definitionerna av orden är helt enkelt inte tillräckligt långa, och många uppenbara kopplingar görs inte. Till exempel klarade inte mitt program att disambiguera det klassiska ordparet ”pine cone”, på grund av att den relevanta betydelsen av ”cone” inte alls fanns med i WordNet. Ordparet ”cat dog” gav också ett oväntat resultat, nämligen definitionerna ”an informal term for a youth or man” och ”informal term for a man”, respektive. Ett av få ordpar som disambiguerades enligt mina förväntningar var ”bank” och ”funds” som gav resultatet ”a financial institution that accepts deposits and channels the money into lending activities” och ”a reserve of money set aside for some purpose”. Jag tänker därför att det är rimligt att dra en liknande slutsats som för min sammanställning ovan, nämligen att bristen på digitaliserad kunskap i ett maskinläsbart format är det största problemet för att kunna skapa välfungerande WSD, och det är på detta problem vi bör sätta in våra resurser för att lösa.

Referenser

Agirre, E. and Edmonds, P. (2007). *Word Sense Disambiguation - Algorithms and Applications*, Springer.

Bird, S., Klein, E. and Loper, E. (2009). *Natural Language Processing with Python*, O'Reilly Media, Sebastopol, California.

Indurkhya, N. and Damerau, F. J. (2010). *Handbook of Natural Language Processing*, second edn, CRC Press, Boca Raton, Florida.

Locke, W. N. and Booth, A. D. (1955). Machine translation of languages: Fourteen essays. Utdrag tillgänglig på http://www.hutchinsweb.me.uk/MTNI-22-1999.pdf.

Internetkällor

- http://wordnet.princeton.edu/
- http://en.wikipedia.org/wiki/SemEval
- http://python.org/
- http://nltk.org/
- http://stackoverflow.com/
- http://www.sussex.ac.uk/Users/robk/OpenMind/template/english/index.html

A Källkoden

```
#!/usr/bin/python

import nltk
from nltk.corpus import wordnet as wn

def lesk_me(word1, word2):

    # First, we ask WordNet to provide the different senses of each word
    senses1 = wn.synsets(word1)
    senses2 = wn.synsets(word2)

    # This checks to see if any of the lists of senses are empty. If so,
    # one or both of the words weren't present in WordNet, and we exit
    # the function.
    if senses1 == [] or senses2 == []:
        print "It seems that one or both of the words are not in the
            dictionary."
        return

    # Here we create an empty dictionary to hold all combinations of senses
    # and a score for the number of words in common.
    overlap_dict = {}

    # This is a list of function words that should be ignored by the
    # algorithm.
    ignorewords = set(['a', 'an', 'the', 'for', 'in', 'to', 'of', 'that', '
        with', 'or', 'and', 'by', ';', ':', '(', ')'])

    # Here we add the definitions to the dictionary, as well as tokenize
    # them and count the number of words they have in common. The
    # definitions are stored as tuple pairs, one for each sense
    # combination, and the count of the number of words in common is
    # stored as a value for each pair.
    for sense1 in senses1:
        for sense2 in senses2:
            tokens1 = set(nltk.word_tokenize(sense1.definition))
            tokens2 = set(nltk.word_tokenize(sense2.definition))
            overlap_dict[sense1.definition, sense2.definition] = len(
                tokens1.intersection(tokens2).difference(ignorewords))

    # Here, we extract the number of the highest score that any sense
    # combination received. That is, the most number of words in common
    # with words from the other sense's definition.
    best_in_dict = max(overlap_dict.values())
```

```
42
43      # If the highest score is 0, we apologize that we couldn't be of
44      # any help.
45      if best_in_dict == 0:
46          print "\nI'm sorry, but the definitions of these words have no
                words in common. I can therefore not help you disambiguate."
47          return
48
49      # Then we make a list of all different definitions that received
50      # the highest score.
51      best_sense_combos = [sense_combo for sense_combo, count in overlap_dict
            .items() if count == best_in_dict]
52
53      # If only one sense received the highest score, we tell what the most
54      # likely definition is, otherwise, we list all the definitions that
55      # received the highest score
56      if len(best_sense_combos) == 1:
57          print '\nFor the word "{0}", the most likely definition is: "{1}".'
                .format(word1, best_sense_combos[0][0])
58          print 'For the word "{0}", the most likely definition is: "{1}".'.
                format(word2, best_sense_combos[0][1])
59      else:
60          possible_defs1 = set([def1 for def1, def2 in best_sense_combos])
61          possible_defs2 = set([def2 for def1, def2 in best_sense_combos])
62          if len(possible_defs1) == 1:
63              print '\nFor the word "{0}", the most likely definition is:
                    "{1}".'.format(word1, best_sense_combos[0][0])
64          else:
65              print '\nI was unable to find a definite match for "{0}". The
                    possible definitions are: '.format(word1)
66              for definition in possible_defs1:
67                  print '* "' + definition + '"'
68          if len(possible_defs2) == 1:
69              print '\nFor the word "{0}", the most likely definition is:
                    "{1}".'.format(word2, best_sense_combos[0][1])
70          else:
71              print '\nI was unable to find a definite match for "{0}". The
                    possible definitions are: '.format(word2)
72              for definition in possible_defs2:
73                  print '* "' + definition + '"'
74
75
76  def main():
77
78      # This is the main function. Here we just ask the user to input two
79      # words and pass them over to the Lesk algorithm.
```

```
80      word1 = raw_input("Enter a word: ").strip()
81      word2 = raw_input("OK, good. Now, enter another word: ").strip()
82      lesk_me(word1, word2)
83
84
85  if __name__ == '__main__':
86    main()
```